***Business rules language specifications***

|  |  |
| --- | --- |
| **Subject** | Business rules language |
| **Date** | 24/02/2009 |
| **Authors** | Jérôme BECK (THALES) |
| **Version** | V1.2 |
| **Reviewers** | Charles-Eric Laporte (THALES) |
|  | Lyazid SABRI, Gian Piero ZARRI (LISSI Laboratory, Univ. Paris 12) |

***Keywords***

[Keywords]

***Abstract***

[Abstract]

Table of content

[I. Introduction 4](#_Toc223407646)

[II. Rules document 4](#_Toc223407647)

[A. Using the micro-concept model 4](#_Toc223407648)

[III. Variables 5](#_Toc223407649)

[A. Introduction 5](#_Toc223407650)

[B. Global variables declaration 6](#_Toc223407651)

[C. Local variables in rules 6](#_Toc223407652)

[IV. Rule language 6](#_Toc223407653)

[A. General form 6](#_Toc223407654)

[B. Rule activation model 7](#_Toc223407655)

[C. Attributes 7](#_Toc223407656)

[1. Priority 7](#_Toc223407657)

[D. Condition 7](#_Toc223407658)

[1. Base pattern 7](#_Toc223407659)

[2. Combining base patterns 11](#_Toc223407660)

[E. Consequence 13](#_Toc223407661)

[1. Property value reading 13](#_Toc223407662)

[2. Instance creation 14](#_Toc223407663)

[3. Instance removal 14](#_Toc223407664)

[4. Property value update 14](#_Toc223407665)

[5. Property value addition 15](#_Toc223407666)

[6. Property value removal 15](#_Toc223407667)

[7. Action on an instance 16](#_Toc223407668)

[8. Variable update 16](#_Toc223407669)

[V. Queries 16](#_Toc223407670)

[A. Parameters 17](#_Toc223407671)

[B. Condition 17](#_Toc223407672)

[VI. Literal types 17](#_Toc223407673)

[A. Integers 17](#_Toc223407674)

[B. Decimals 18](#_Toc223407675)

[C. Booleans 18](#_Toc223407676)

[D. String 18](#_Toc223407677)

[E. Date, time, datetime 18](#_Toc223407678)

[F. Time duration 19](#_Toc223407679)

[VII. Language reference 19](#_Toc223407680)

[A. Keywords list 19](#_Toc223407681)

# Introduction

Business rules will be one of the core elements of the reasoning service in SEMbySEM. The main requirement of the rule-model we propose is to be able to express rules in term of micro-concept vocabulary and to address instances of this model. The base mechanism for this type of reasoning is:

*If condition (instances pattern) then consequence (operations on instances).*

Once the properties updates from the façade are integrated in the core system, we start the business rules engine to find any condition that is verified, and apply the associated consequences. Each operation may modify instances, which will validate a new rule and etc.

We need a reasoner able to handle efficiently a model constantly in change. For this purpose, production rules engines based on the RETE algorithm seem particularly adapted. RETE based reasoner always have their own data model that have some similarities with the micro-concept model but that don’t allow us to use the micro-concept model completely. For this reason, we need a language that allows some constructions usually not handled by business rules languages, like handling properties with multiple values.

Moreover, this language is an attempt to provide a way to edit rules as natural and easily as possible. We don’t provide every feature available is RETE based engines, but only functionalities that are useful and simple enough for the user.

# Rules document

A rule document is constituted of three sections:

* Ontologies declaration
* Global variables declaration
* Rules

Any part of the document can contain comments. We provide two types of comments:

* Single line comments, starting with a ‘//’. Every character after this double slash until the end of the line is in the comment.

Ex:

//Comment

* Multiline comments, enclosed between a ‘/\*’ and a ‘\*/’

Ex:

/\*Comment line 1

Comment line 2\*/

## Using the micro-concept model

References to micro-concept elements must be done enclosing the element ID inside simple quotes. If a quote or a back-slash is needed in this section, it must be escaped by a back-slash. The other characters can’t be escaped; doing so will be interpreted as a syntax error.

Ex:

Concepts: ‘Transportation/Train’, ‘Transportation/MyStation’

Properties: ‘Transportation/HasWagon’

Instances: ‘Transportation/TGV1234’

Actions: ‘Transportation/Train/EmergencyStop’

Enumerations: ‘Network/ShutdownReason’

Enumeration values: ‘Network/ShutdownReason/Maintenance’

To accelerate the authoring of rules and enhance its edition, we allow two operations:

* Ontology declaration: the user can specify the name of ontologies used in the rest of the document. These declarations must be placed in the global declarations of the document. The elements within this ontology can be referenced without writing the name of their ontology if the resulting reference isn’t ambiguous.

Ex:

using ontology ‘Transportation’;

In this case, elements of this ontology can be used in rules without the ‘Transportation’ suffix:

‘Train’, ‘MyStation’, ‘HasWagon’

* If the micro-concept element is written with the syntax [a-zA-Z\_][ a-zA-Z0-9\_]\*, then the quotes are optional.

Ex:

using ontology Transportation;

And in rules:

Train, MyStation, HasWagon

# Variables

## Introduction

Variables can be declared in a rules document to store instances, properties values or literal values. Variables are always explicitly declared to avoid errors while using them. When variables are used without being declared, there is no way to see variables written with a typo, which can lead to bug difficult to detect.

Moreover, variables in our representation are always typed to benefit the security provided by a strong typing. The type of the variable must be given when the variable is declared.

There is an exception to these rules: a variable used to assign a value of a property value or of a pattern matched in a rule’s condition doesn’t need to be explicitly declared, and its type will be automatically deduced from the matched element.

A variable name is formed with alphanumerical characters or underscores characters with a starting ‘?’. The first character after the ‘?’ must be a letter or an underscore. We force a starting ‘?’ to prevent name ambiguities between variables and short micro-concept elements names. In particular, without this syntax, if the micro-concept model was modified and if a new concept was created with the same name as a variable in the document, the document would not compile anymore.

<Variable Name> ::= ‘?’[a-zA-Z\_][a-zA-Z0-9\_$]\*

## Global variables declaration

The global declarations part can be used to declare global variables, available in every rule. The type and the name of the global variable must be declared. The initial value of the variable can also be given, but this is optional.

Ex:

integer ?thresholdValue := 5;

Train ?selectedTrain;

Station ?myStation := ‘MyStation’;

## Local variables in rules

Variables used in a rule’s condition or consequence part are available in the rest of the rule, but not for other rules. However, if a variable is used in a ‘not’, ‘exists’ or ‘forall’ section, the variable is not visible outside of this section.

There can’t be two variables with the same name available in the same scope, even if they are declared in different scopes. In particular, it’s not possible to declare a local variable that has the same name as a global variable. We impose those strict rules to avoid classical errors caused by this practice.

**Condition part:**

Every variable created in this part is used to assign the value of a matched instance or of a property value. As explained in the introduction, those variables are not declared explicitly. Each assignment is an implicit variable declaration, and the variable will have the type of the matched element. In particular, this means that the variable used must not be already declared.

**Consequence part:**

Variables in this part must be declared like global variables. They can also be assigned in the declaration statement, but this is not mandatory.

# Rule language

This part explains the complete syntax supported to express a rule.

## General form

Each rule will have the global form:

Rule "ruleName"

[Attributes]

If

{

[Condition]

}

Then

{

[Consequence]

}

## Rule activation model

The rule will be activated the first time the implied instances verify the conditions, and every time the properties on the instances change (if the condition still remains true).

Ex: Rule = If Train where speed > 100, then (do something).

The speed property has the following values: 30,120, 90, 70, 150, 180.

The rule engine will do:

Nothing, do, nothing, nothing, do, do.

## Attributes

### Priority

If two rules can be applied at the same time, the rule engine will execute the one with the highest priority first.

The user can write a priority value for each rule:

Ex:

priority := 10;

Ex:

priority := -50;

## Condition

In this section, we detail the part constituting a rule’s activation condition.

### Base pattern

#### Matching with functional properties constraints

The base part of a rule’s condition is to match a micro-concept instance:

Ex:

Train() //Match every instance of Train

The rule can declare that the instance must follow some properties restriction. The properties restrictions are located inside the match parenthesis, separated with commas.

Train(Speed > 300, Color == "blue")

If a property value has to be used in another matched pattern or in the rules consequence, the property value can be assigned in a variable:

Train(?speed := Speed)

The assigned variable can also be used in the current pattern:

Train(?s := Speed, ?s > 300)

The instance matched can also be assigned in a variable to be resued in another pattern or in the rules consequence:

train := Train(Speed > 300, Color == "blue")

Remark: the rule will be reactivated everytime the value of a property in the pattern constraints changes. This is also true for a property value that is assigned to a variable, even if this variable isn’t resued in the consequence part. This can be an advantage or a drawback depending on the real purpose of the rule.

#### Set/List properties constraints

If the property checked is a property with a multiple cardinality, two options are available:

* Either match every value of this property individually.

Ex:

Train(?wagon := one(HasWagon)),

?wagon(Weight > 10)

In this case, the rule will be activated for every train, and for every wagon within this train whose weight is > 10.

* Or match every value of this property globally. The matched property can be used in the operations ‘not’, ‘exists’, and ‘forall’ to check a constraint on every value of a property.

Ex: Look every train containing at least one wagon whose weight is > 10:

Train(?wagons := all(HasWagon)),

exists(one in ?wagons(Weight > 10))

The rule is activated for every train matching the condition but not for every wagon within this train.

Notice that in this case, we use a different syntax to match the wagons list (Wagon in wagons). This syntax must be used if and only if the variable contains a list of properties values, obtained with the all() operator.

Ex: Look every train that doesn’t contain any wagon whose weight is > 10:

Train(?wagons := all(HasWagon)),

not(one in ?wagons(Weight > 10))

The variable can be used without a ‘not’, ‘exists’ or ‘forall’ operator. This example is equivalent to the first example of this section:

Train(?wagons := all(HasWagon)),

one in ?wagons(Weight > 10)

However, the ‘all’ keyword should be used to test a property globally with a ‘not’, ‘exists’ or ‘forall’ operator. The following syntax is valid, but isn’t right:

Train(?wagon := one(HasWagon)),

exists(?wagon(Weight > 10))

//Match every train and every wagon such as the wagon has a weight > 10.

//The ‘exists’ is useless here.

#### Cardinality restriction

The condition can also be a restriction on a property cardinality (available only for properties with multiple cardinality):

Train(count(HasEngine)==0)

Train(count(HasEngine)<count(HasWagon))

The cardinality of the property can be stored in a variable if needed:

Train(?numWagons := count(HasWagon))

#### Property restriction syntax

A property restriction is a comparison of two expressions. The comparison operator can be:

== (equals), != (not equals), < (lower), <= (lower or equals), > (greater), >= (greater or equals)

Each comparison operand is an expression that may be:

* A property of the matched instance.
* A literal value.
* A variable (either global or local to the rule).
* A static instance.
* An expression enclosed between parentheses.
* An arithmetic operation using the operator (+, -, \*, /, % (modulus)) and two expression operands, except for the minus operation that may have only one expression operand. The operation is written with the usual mathematical notation and operations priorities: + and – have the lowest priority, \*, / and % have the highest priority.
* Those operations are only available for literal values or for properties or variables with a literal value. Here is the details on the operations supported for each type:
  + Integers supports: +, -, \*, /, %
  + Decimals supports: +, -, \*, /
  + For date/time, it’s possible to do:
    - date/time +,- time span -> date/time
    - date/time - date/time -> time span
  + Strings and enumerations don’t support those operations

The calculation operators follow the usual priorities: The ‘equals’ or ‘not equals’ comparison operations are available for any expression (including properties or variables with a value that is a micro-concept instance).

The inequality comparisons are only available for expressions with a literal value that is either:

* An integer
* A decimal
* A date/time
* A time span

Ex:

Train(-Speed\*5>1000/Weight+(50\*count(HasWagon))%3)

Remark: It’s not possible to assign an expression with this form. Only properties values and properties cardinality can be assigned into a variable.

#### Property restrictions combination

If an instance matching has several properties constraints, they are linked with an implicit ‘and’ operator. It’s also possible to use explicitly ‘and’ and ‘or’ operators to combine properties constraints.

Ex :

Train(Speed>100 or Weight>10)

Train(Platform == platform1, (ArrivesAt > time1 and ArrivesAt < time2)

or (LeavesAt > time1 and LeavesAt < time2))

#### Instance matching

**Single instance matching:**

Instead of matching any instance of a certain type, it’s possible to specify the exact instance to match. The instance can be directly written with its ID value, or it can be given with a variable containing its value.

Ex :

?train3452(Speed < 150)

In this example, we trigger the rule only if our particular variable containing an instance of train has a speed <150. In this case, if a variable is used, it’s possible to qualify the type expected for the contained instance.

Ex :

TGV ?train3452(Speed < 150, FloorsNumber == 2)

The rule will only trigger if the variable train3452 is a TGV (and not only a Train). With this particular qualification, we are able to use the property ‘FloorsNumber’, which is only available for a TGV and not for a simple Train.

If a static instance is used in the match, it will automatically be used in its type, and it can not be qualified any further. If a variable is used, by default, the type used will be:

* The type specified in the declaration of the variable for a global variable.
* The type of the matched data (micro-concept or property type) if it’s a local variable.

In this case, it’s possible to assign the qualified instance in a new variable.

Ex:

?tgv := TGV ?train3452(Speed < 150, FloorsNumber == 2)

**Multiple instance matching:**

If a variable was created by assigning a property with multiple cardinality, using the ‘all’ keyword, the variable can be resued to match the instances within the variable. In this case, the syntax is different.

Ex:

one in ?wagons(Weight > 10)

In this case, the type of concept used for the match will be the range of the property that was assigned to this variable.

The matching type can be overconstrained as before, and in this case, the match will be activated for every value in the variable whose type is a sub-concept of the given concept.

Ex:

PassengerWagon in ?wagons(Weight > 10)

In this case, it’s possible to assign the qualified instance in a new variable.

Ex:

?pWagon := PassengerWagon in ?wagons(Weight > 10)

### Combining base patterns

#### And

The condition part can contain several base or composed patterns, separated by a comma. If several patterns are placed in the condition part, they are linked with an implicit ‘and’ operator. The commas can be replaced by explicit ‘and’ operators.

Ex:

t1:=Train( ?pos:=Position),

t2:=Train(Position == ?pos)

or

t1:=Train( ?pos:=Position) and

t2:=Train(Position == ?pos)

It’s possible to check a condition on the property value of a match instance, cascading two instance matching.

Ex:

?t:=Train( ?wagon:=HasWagon, Speed>100)

wagon(Weight > 10)

Here we match every couple of (Train,Wagon) such as the train has a speed > 100, and the wagon belongs to this train and has a weight > 10.

#### Not

The condition is true if the internal condition isn’t.

Ex:

not(Train(Speed>100))

The rule is activated if no train has a speed > 100.

not(

?t1:=Train( ?pos:=Position) and

?t2:=Train(Position == ?pos, isnot( ?t1))

)

The rule is activated if there isn’t a couple of trains with the same position. In this case, we need an explicit ‘and’ to link the base patterns. Note that the variables inside the ‘not’ (t1, t2, …) are not available outside of the scope of ‘not’.

#### Exists

The rule is activated if the internal condition is validated at least once. If several instances validate the condition, the rule is only activated once. This is equivalent to a ‘not(not())’ expression. The variables inside the ‘exists’ are not available outside of the scope of ‘Exists’.

Ex:

exists(Train(Position == MyStation))

The rule is activated if there is at least one train in ‘MyStation’

#### Forall

A rule can use this operator with the form:

Forall (combined pattern; combined pattern).

The rule will be activated if for every instance or instances n-uple checking the first pattern, the following condition will be validated for at least one instance. This is equivalent to:

Not (‘combined pattern’ And Not(‘combined pattern’))

(there is(are) no instance(s) such as the first pattern is true and the second pattern is false)

Ex:

forall ( ?t:=Train(Position == MyStation); Train ?t(Speed == 0))

The rule is activated if every train in ‘MyStation’ is stopped.

Ex:

forall (?s:=Station(count(Platform)>10) and ?t:=Train(Position == ?s); ?t(Speed == 0))

The rule is activated if for every station that has more than 10 platforms and for every train in the station, the train is stopped.

Ex:

?g:=DetectorGroup() and forall(?d:=Detector(BelongTo == ?g); Detector ?d(Status == Danger))

The rule is activated for every detector group whose detectors all signal a danger.

#### Is Not

If several objects are match in a condition, those objects can actually reference the same instance. In some cases, we need to explicitly declare those objects as different.

?t1:=Train( ?pos:=Position),

?t2:=Train(Position == ?pos)

In this example, the rule will be activate for each train, t1 and t2 referencing this same train. In this case, we want the two matched object to be different. We need additional parameter to activate the rule only if the two instances are different.

Ex:

?t1:=Train( ?pos:=Position),

?t2:=Train(Position == ?pos, isnot( ?t1))

#### Conditions outside of a pattern

It possible to write a condition that must be checked in the rule outside of a pattern. The syntax allowed is the same as properties constraints in a pattern:

* Expressions with arithmetic operators.
* Comparison operators.
* And/Or operators.

Ex: Find two different trains with one wagon in common or with two wagons whose added weight is greater than 20

?t1:=Train(?wagons1 := all(HasWagon)),

?t2:=Train(?wagons2 := all(HasWagon), isnot(?t1)),

not(exists(

?wagon1 := one in ?wagons1(?weight1 := Weight),

?wagon2 := one in ?wagons2(?weight2 := Weight),

?wagon1 == ?wagon2 or (?weight1+?weight2)>20

)

## Consequence

As described in the business rules model document, the following operations are allowed in the consequence part of a rule:

* Instance creation/removal
* Property value update
* Property value creation/removal
* Action on an instance
* Global variable update

The consequence part can define several operations that need to be done when the rule is activated. Each instruction is ended by a semi-column.

### Property value reading

Properties of values containing a micro-concept instance can be accessed directly in the consequence part of a rule. The base micro-concept instance can be:

* An instance ID.
* A variable containing an instance.
* A property value that is an instance, obtained with the current construction.

The property read on this instance must be a functional property.

Ex:

Train ?train := ?platform->HasTrain;

We read the value of the property ‘HasTrain’ of the instance contained in the platform variable.

It’s possible to access to the value of a property value of a property in a single call, as long as the two implicated properties are functional. In this case, the first property must have a range that is a concept (and not a literal type). The following syntax is valid:

?number := ?platform->HasTrain->TrainNumber;

Remark: The value of a functional property can be ‘null’. In this case, to simplify the authoring of rules, if a sub property of this parent property is asked, it will also return null.

### Instance creation

The instance is created using the ‘createInstance’ keyword, specifying the name of the micro-concept to create. The created instance can optionally be stored in a variable to be used in a following operation.

Remark: the system automatically assigns the ID of the created instance.

Ex:

createInstance(FireAlarm);

Train ?newTrain;

?newTrain:=createInstance(Train);

### Instance removal

An instance is removed using the ‘removeInstance’ keyword, specifying either:

* A variable containing the instance to remove
* The ID of the instance

Ex:

removeInstance(?oldTrain);

removeInstance(‘Transportation/tgv4567’) //This syntax is supported, though it might not be very useful to remove a static instance

### Property value update

A property value is modified specifying the instance containing the property, and the property name. The instance can be referenced directly with an ID or with a variable. However, it’s not possible to declare the property value to change using a variable that contains a property value. The assigned value can be a variable, a literal value or an ID, and must be compatible with the type of the property. It can also be an expression obtained with arithmetic operations.

This feature is only available for functional properties.

Ex:

?train1->TrainNumber := ?calculatedNumber;

?train1->’Transportation/Train/Speed’ := ?train1->HasEngine->Speed;

### Property value addition

A property value is added to an instance specifying the instance containing the property, and the property name. The instance can be referenced directly with an ID or with a variable. However, it’s not possible to declare the property value to change using a variable that contains a property value. The added value can be a variable, a literal value or an ID, and must be compatible with the type of the property. It can also be an expression obtained with arithmetic operations.

This feature is only available for non functional properties.

Ex:

addPropertyValue(?train1->HasWagon, ?wagonInStock);

addPropertyValue(?train1->HasWagon, ‘Transportation/MyStation/Wagon9875’);

For properties with ordered values, it’s possible to specify the position where the value must be inserted.

Ex:

insertPropertyValue(?train1->HasWagon, ?wagonInStock, 0);

insertPropertyValue(?train1->HasWagon, {Transportation/MyStation/Wagon9875}, 0);

For this instruction, the last argument defines the position where the value is placed, the current values with a position greater than or equals to this value will be placed after this element. This instruction can be used to place a value as the first value of the property, as demonstrated in our sample. The given position must be either a variable or a literal (integer), or an expression constituted of arithmetic operations.

It the addPropertyValue instruction is used on a property with ordered values, then the value will be added at the last position.

### Property value removal

A property value is removed from an instance specifying the instance containing the property, and the property name. The instance can be referenced directly with an ID or with a variable. However, it’s not possible to declare the property value to change using a variable that contains a property value. The removed value can be a variable, a literal value or an ID, and must be compatible with the type of the property.

This feature is only available for non functional properties.

Ex:

removePropertyValue(?train1->HasWagon, ?wagonInStock);

removePropertyValue(?train1->HasWagon ,’Transportation/MyStation/Wagon9875’);

If the instance has the property value to be removed several times, this instruction will only remove the first one found with this value.

For ordered properties, it’s possible to specify the position of the property to remove. The given position must be either a variable or a literal (integer), or an expression constituted of arithmetic operations.

Ex:

removePropertyValueAt(?train1->HasWagon, 0);

removePropertyValueAt(?train1->HasWagon, -1); //remove the last element

It’s also possible to remove every value associated to a property.

Ex:

removeAllPropertyValues(?train1->HasWagon);

### Action on an instance

Actions can be executed directly if they don’t need any input or output parameters. Otherwise, an action parameter must be created first using the ‘createAction’ instruction. The created parameter can be used to set input parameters, or retrieve output parameters after the action execution.

Ex:

execute( ?train1,’Transportation/Train/EmergencyStop’);

‘Network/Actions/Shutdown’ ?shutdownAction := createAction(?proxyServer, ‘Network/Actions/Shutdown’);

?shutdownAction->Reason := ‘Network/ShutdownReason/Maintenance’;

execute(?shutdownAction);

### Variable update

Variable values can be changed, simply by assigning a new value to them. This operation is available for both local and global variables. This new value can be a literal value, a static instance, a variable or an expression obtained with arithmetic operations.

Ex:

?train := trainInStation;

?thresholdValue := 2;

# Queries

Each rule will have the global form:

Query "queryName"

[Parameters]

If

{

[Condition]

}

## Parameters

Queries can define input and output parameters. Each parameter variable must be declared with its type.

The input parameters specify variables defined by the user of the query that can customize the query.

The output parameters define the variables assigned in the condition expression that will be transferred to the user of the query as its result.

Ex:

Query "myQuery"

Input Train ?train;

Input integer ?limitWeight;

Output Wagon ?wagon;

If

{

?train(?wagon := one(HasWagon))

?wagon(Weight>?limitWeight)

}

This query returns every wagon that belongs to the input train whose weight is greater than the input weight threshold (stored in limitWeight).

## Condition

The expression of the condition in a query is exactly the same as the expression of the condition in the rule.

# Literal types

We remind here the list of types supported by our model:

* Integer
* Decimal
* Boolean
* String
* Date, time, datetime
* Time duration

## Integers

An integer can be declared using the ‘integer’ keyword. For efficiency matters, we store those values in native 64 bits integer values, which means that their value must be within the range:

-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Ex:

integer ?threasholdValue := 5;

## Decimals

A decimal can be declared using the ‘decimal’ keyword. For efficiency matters, we store those values in native 64 bits floating point values, following the IEEE standard for floating point values. The value will have the approximate range: ±x.10e

1≤x<10, x being a decimal number with 15 or 16 significant digits

-324≤e≤308

The number can be written directly as a decimal number or using a scientific notation with an exponent.

Ex:

decimal ?a := -124.657;

decimal ?a := -1.24657e+2;

decimal ?b := 0.0025;

decimal ?b := 2.5e-3;

## Booleans

A Boolean is declared using the ‘boolean’ keyword. The value can only take two values: ‘true’ and ‘false’.

Ex:

boolean ?state1 := true;

boolean ?state2 := false;

## String

A string can be declared using the ‘string’ keyword.

Ex:

string ?text := "my text";

To write double quotes or backslashes inside a string, those characters must be escaped with a backslash. Other characters can’t be escaped and it will generate an error if this case is detected. We impose this constraint to avoid the ambiguity when interpreting a single backslash.

Ex:

string ?text := "text with \" and \\ inside" -> text with " and \ inside

string ?text := "te\xt" -> error

## Date, time, datetime

Dates and time are written as a string in a format compatible with the XSD date and time format (including the handling of timezones).

A date has the form “YYYY-MM-DD” but it’s not mandatory to use 2 digits to represent the month or the day. The year must be written completely.

Ex:

date ?mydate := date("2009-01-21");

date ?mydate := date("2009-1-21");

A time will have the form “HH:MM:SS”. The seconds section can be decimal to define fractions of seconds. We support a precision of 1ms. Once again, the number of digits is not imposed.

Ex:

time ?mytime := time("12:45:15");

time ?mytime := time("8:36:17.47"); //17s470ms

time ?mytime := time("1:4:054");

It’s possible to combine the two concepts in a ‘datetime’ literal. The value starts with the date, use a ‘T’ as separator and then specify the time.

Ex :

datetime ?t:= datetime("2009-01-21T12:45:15");

Remark : the date and time written must be valid. It’s not possible to write a value out of bound and expect the parser to convert the exceeding value into the higher time unit.

Ex :

time("12:65:15") //is invalid

It’s possible to use the current date/time:

datetime("now")

## Time duration

This data type expresses a duration of time. Once again, we use a format compatible with the XSD duration format, except duration with years or month because their signification is ambiguous: [-]PxDTxHxMxS.

Every letter can be written in upper or lower case, and spaces can be placed anywhere to increase the readability of the time duration. The starting ‘P’ is optional. Moreover, since we don’t support month duration, there isn’t an ambiguity in the signification of the ‘M’, and we can make the ‘T’ separating days and hours optional. Seconds can be written with a decimal value, with a precision up to the millisecond.

Ex:

duration ?t := duration("15d"); // 15 days

duration ?t := duration("1d 12h"); // 1 day and 12 h

duration ?t := duration("2.5s"); // 2s500ms

duration ?t := duration("-2.5s"); // 2s500ms

duration ?t := duration("110s"); // equivalent to 1mn50s

# Language reference

## Keywords list

Here is the list of keywords reserved by the language that can’t be used as variable names. Micro-concept elements can use those names if they are enclosed in simple quotes.

Structure:

* Rule, Query, If

Rule content:

* Priority, Input, Output, and, or, not, exists, forall, count, one, all, isnot, in

Types:

* integer, decimal, boolean, string, date, time, datetime, duration, true, false